Where we are

- Done: Syntax, semantics, and equivalence
  - As long as all you have is loops and global variables
- Now: Didn’t IMP leave some things out?
  - Particularly scope, functions, and data structures
  - (Not to mention threads, I/O, exceptions, strings, ...)

Time for a new model... (Pierce, chapter 5)
Data + Code

Higher-order functions work well for scope and data structures.

- Scope: not all memory available to all code
  
  let x = 1
  let add3 y =
      let z = 2 in
      x + y + z
  let seven = add3 4

- Data: Function closures store data. Example: Association “list”
  
  let empty = (fun k → raise Empty)
  let cons k v lst = (fun k’ → if k’=k then v else lst k’)
  let lookup k lst = lst k

(Later: Objects do both too)
Adding data structures

Extending IMP with data structures isn’t too hard:

\[
\begin{align*}
e &::= c \mid x \mid e + e \mid e \cdot e \mid (e, e) \mid e.1 \mid e.2 \\
v &::= c \mid (v, v) \\
H &::= \cdot \mid H, x \mapsto v
\end{align*}
\]

\[
H; e \downarrow c
\]

\[
\[
\begin{array}{cccc}
    & H; e_1 \downarrow v_1 & H; e_2 \downarrow v_2 & H; e \downarrow (v_1, v_2) & H; e \downarrow (v_1, v_2) \\
\cdots & H; e_1 \downarrow v_1 & H; e_2 \downarrow v_2 & H; e \downarrow (v_1, v_2) & H; e \downarrow (v_1, v_2) \\
    & H; (e_1, e_2) \downarrow (v_1, v_2) & H; e.1 \downarrow v_1 & H; e.2 \downarrow v_2 & \\
\end{array}
\]

Note: We allow pairs of values, not just pairs of integers

Note: We now have stuck programs (e.g., \( c.1 \)) – what would C++ do? Scheme? ML? Java? Perl?

Note: Division also causes stuckness
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
  e & ::= \ldots \mid \text{fun } x \rightarrow s \\
  s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
\begin{array}{c}
  H; e \downarrow c \\
  H; s \rightarrow H'; s'
\end{array}
\]

\[
\begin{array}{c}
  H; \text{fun } x \rightarrow s \downarrow \text{fun } x \rightarrow s \\
  H; e_1 \downarrow \text{fun } x \rightarrow s
\end{array}
\begin{array}{c}
  H; e_2 \downarrow v \\
  H; e_1(e_2) \rightarrow H; x := v; s
\end{array}
\]

Does this match “the semantics we want” for function calls?
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
  e & ::= \ldots \mid \text{fun } x \to s \\
  s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
\begin{array}{c}
\text{H;e}_1\downarrow\text{fun } x \to s \\
\text{H;e}_2\downarrow v
\end{array}
\]

\[
\frac{
\text{H;fun } x \to s\downarrow\text{fun } x \to s}
{\text{H;fun } x \to s\downarrow\text{fun } x \to s}
\]

\[
\frac{
\text{H;e}_1(e_2) \to H \mid x := v; s}
{\text{H;e}_1(e_2) \to H \mid x := v; s}
\]

NO: Consider \( x := 1; (\text{fun } x \to y := x)(2); \) \( \text{ans} := x. \)

Scope matters; variable name doesn’t. That is:

- Local variables should “be local”
- Choice of local-variable names should have only local ramifications
Another try

\[
\frac{H; e_1 \downarrow \text{fun } x \rightarrow s \quad H; e_2 \downarrow v \quad y \ “\text{fresh}”}{H ; e_1(e_2) \rightarrow H ; y := x; x := v; s; x := y}
\]

- “fresh” isn’t very IMP-like but okay (think malloc)
- not a good match to how functions are implemented
- yuck
- NO: wrong model for most functional and OO languages
  (even wrong for C if \( s \) calls another function that accesses the global variable \( x \))
The wrong model

\[ H; e_1 \downarrow \text{fun x -> s} \quad H; e_2 \downarrow v \quad y \text{ “fresh”} \]
\[ H ; e_1(e_2) \rightarrow H ; y := x; x := v; s; x := y \]

\[ f_1 := (\text{fun x -> } f_2 := (\text{fun z -> ans := x + z})); \]
\[ f_1(2); \]
\[ x := 3; \]
\[ f_2(4) \]

“Should” set ans to 6:
- \( f_1(2) \) should assign to \( f_2 \) a function that adds 2 to its argument and stores result in \( \text{ans} \).

“Actually” sets \( \text{ans} \) to 7:
- \( f_2(2) \) assigns to \( f_2 \) a function that adds the current value of \( x \) to its argument.
Punch line

The way higher-order functions and objects work is not modeled by mutable global variables. So let’s build a new model that focuses on this essential concept (can add other IMP features back later).

(Or just borrow a model from Alonzo Church.)

And drop mutation, conditionals, integers (!), and loops (!)

The Lambda Calculus:

\[
e ::= \lambda x. \ e \ | \ x \ | \ e \ e \\
v ::= \lambda x. \ e
\]

You apply a function by *substituting* the argument for the *bound variable*.

(There’s an equivalent *environment* definition not unlike heap-copying; see future homework.)
Example Substitutions

\[ e ::= \lambda x. e \mid x \mid e \, e \]

\[ v ::= \lambda x. e \]

Substitution is the key operation we were missing:

\[ (\lambda x. x)(\lambda y. y) \rightarrow (\lambda y. y) \]

\[ (\lambda x. \lambda y. y \, x)(\lambda z. z) \rightarrow (\lambda y. y \, \lambda z. z) \]

\[ (\lambda x. x \, x)(\lambda x. x \, x) \rightarrow (\lambda x. x \, x)(\lambda x. x \, x) \]

After substitution, the bound variable is gone, so its “name” was irrelevant. (Good!)

There are irreducible expressions \((x \, e)\)
A Programming Language

Given substitution \((e_1[e_2/x])\), we can give a semantics:

\[
e \rightarrow e'
\]

\[
(\lambda x. e) v \rightarrow e[v/x] \quad e_1 e_2 \rightarrow e'_1 e_2 \quad e_2 \rightarrow e'_2
\]

A small-step, call-by-value (CBV), left-to-right semantics

- Terminates when the “whole program” is some \(\lambda x. e\)

But (also) gets stuck when there’s a free variable “at top-level”
(Won’t “cheat” like we did with \(H(x)\) in IMP because scope is what we’re interested in)

This is the “heart” of functional languages like Caml (but “real” implementations don’t substitute; they do something equivalent)
Where are we

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done)
- Notes on concrete syntax
- Simple Lambda encodings (it’s Turing complete!)
- Other reduction strategies
- Defining substitution
Syntax Revisited

We (and Caml) resolve concrete-syntax ambiguities as follows:

1. \( \lambda x. e_1 \ e_2 \) is \( (\lambda x. \ e_1 \ e_2) \), not \( (\lambda x. \ e_1) \ e_2 \)

2. \( e_1 \ e_2 \ e_3 \) is \( (e_1 \ e_2) \ e_3 \), not \( e_1 \ (e_2 \ e_3) \)
   (Convince yourself application is not associative)

More generally:

1. Function bodies extend to an unmatched right parenthesis
   Example: \( (\lambda x. \ y(\lambda z. \ z)w)q \)

2. Application associates to the left
   Example: \( e_1 \ e_2 \ e_3 \ e_4 \) is \( (((e_1 \ e_2) \ e_3) \ e_4) \).
   • These strange-at-first rules are convenient
     • Like in IMP, we really have trees
       (with non-leaves labeled \( \lambda \) or “application”)

Dan Grossman  CSE505 Fall 2009, Lecture 6  13
Simple encodings

Fairly crazy: we left out constants, conditionals, primitives, and data structures

In fact, we’re *Turing complete* and can *encode* whatever we need

Motivation for encodings:

- Fun and mind-expanding
- Shows we aren’t oversimplifying the model
  ("numbers are syntactic sugar")
- Can show languages are *too expressive*
  (e.g., unlimited C++ template instantiation)

Encodings are also just “(re)definition via translation”
Encoding booleans

There are two booleans and one conditional expression. The conditional takes 3 arguments (via currying). If the first is one boolean it evaluates to the second. If it’s the other boolean it evaluates to the third.

*Any 3 expressions meeting this specification (of “the boolean ADT”) is an encoding of booleans.*

“true” \( \lambda x. \lambda y. x \)

“false” \( \lambda x. \lambda y. y \)

“if” \( \lambda b. \lambda t. \lambda f. b \ t \ f \)

This is just one encoding.

E.g.: “if” “true” \( v_1 \ v_2 \rightarrow^* v_1 \).
Evaluation Order Matters

Careful: With CBV we need to “thunk”...

“If” “true” ($\lambda x. \ x$) ($\overbrace{(\lambda x. \ x \ x)(\lambda x. \ x \ x)}$)

an infinite loop

diverges, but

“If” “true” ($\lambda x. \ x$) $\overbrace{((\lambda x. \ x \ x)(\lambda x. \ x \ x))z}$)

a value that when called diverges

doesn’t.
Encoding pairs

The "pair ADT" has a constructor taking two arguments and two selectors. The first selector returns the first argument passed to the constructor and the second selector returns the second.

"mkpair" $\lambda x. \lambda y. \lambda z. z \ x \ y$

"fst" $\lambda p. p(\lambda x. \lambda y. x)$

"snd" $\lambda p. p(\lambda x. \lambda y. y)$

Example:

"snd" ("fst" ("mkpair" ("mkpair" $v_1 \ v_2$) $v_3$)) $\rightarrow^* v_2$
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list is "mkpair" "false" "false"
- Non-empty list is "mkpair" "true" ("mkpair" h t)
- Is-empty is ...
- Head is ...
- Tail is ...

(Not too far from how lists are implemented.)
Encoding natural numbers

Known as “Church numerals” — see the text (or don’t bother).

We can define the naturals as “zero”, a “successor” function, an “is equal” function, a “plus” function, etc.

The encoding is correct if “is equal” always returns what it should, e.g., is-equal (plus (succ zero) (succ zero)) (succ(succ zero)) should evaluate to “true”
Recursion

Some programs diverge, but can we write *useful* loops? Yes!

To write a recursive function:

- Write a function that takes an $f$ and calls it in place of recursion
  - Example (in enriched language):
    $$\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1))$$

- Then apply “fix” to it to get a recursive function:
  - “fix” $\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1))$

- “fix” $\lambda f. e$ will reduce to *something roughly equivalent to* $e[(\text{“fix” } \lambda f. e)/f]$, which is “unrolling the recursion once” (and further unrollings will happen as necessary).

- The details, especially for CBV, are icky; the point is it’s possible and you define “fix” only once

- Not on exam: “fix” $\lambda f. (\lambda x. f (\lambda y. x x y))(\lambda x. f (\lambda y. x x y))$
Where are we

- Motivation for a new model
- CBV lambda calculus using substitution
- Notes on concrete syntax
- Simple Lambda encodings (it’s Turing complete!)
- Next: Other reduction strategies
- Defining substitution